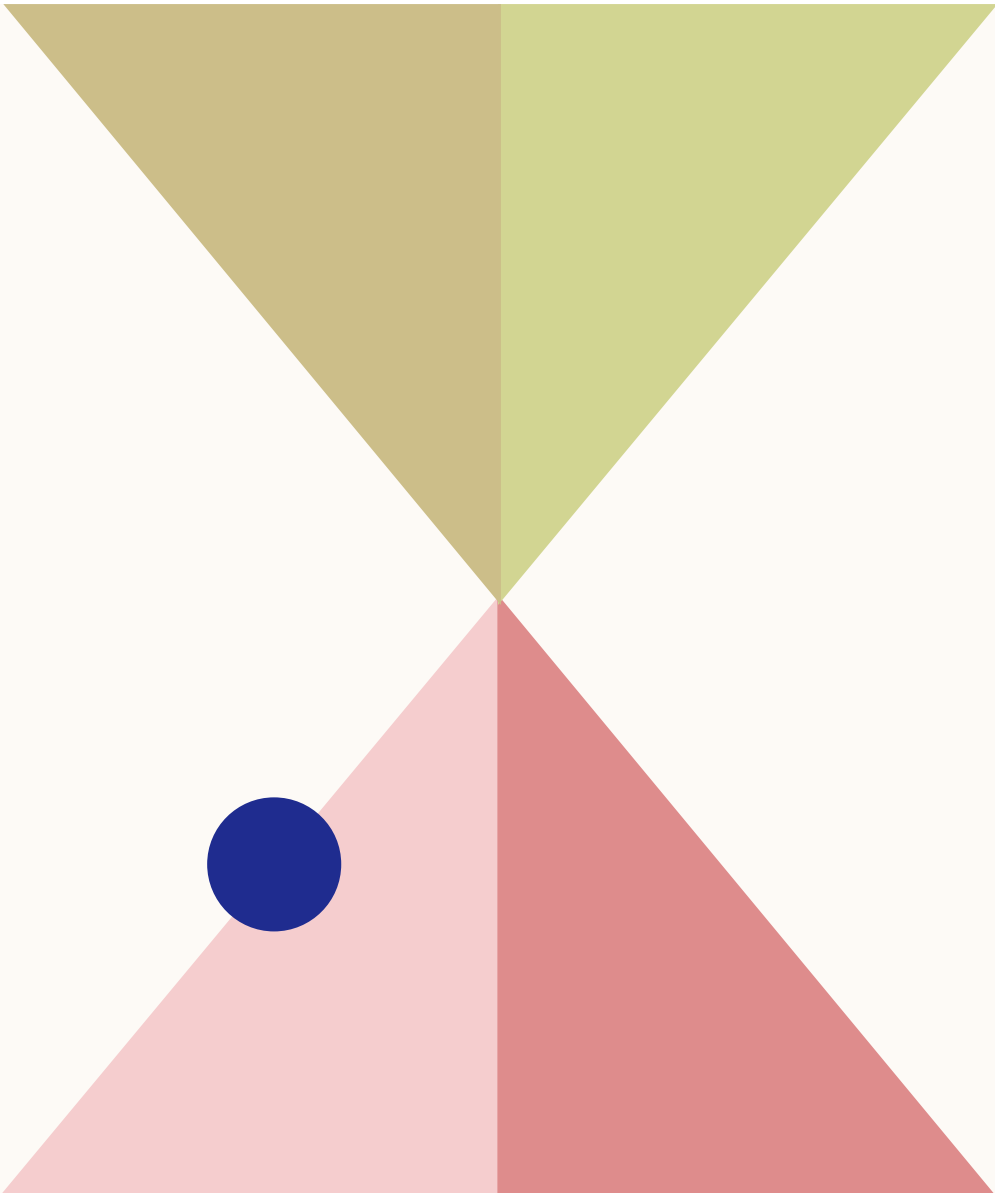
DISTRIBUTED SYSTEM DESIGN

GROUP :16

GROUP MEMBERS

1. Anosh Kurian Vadakkeparampil – 40303184
2. Sahiti Chilakala – 40304091
3. Sushmitha Tiwari Ganga – 40316952
4. Prakash Yuvaraj – 40302061
5. Tharun Balaji - 40312033

INDEX



Introduction:

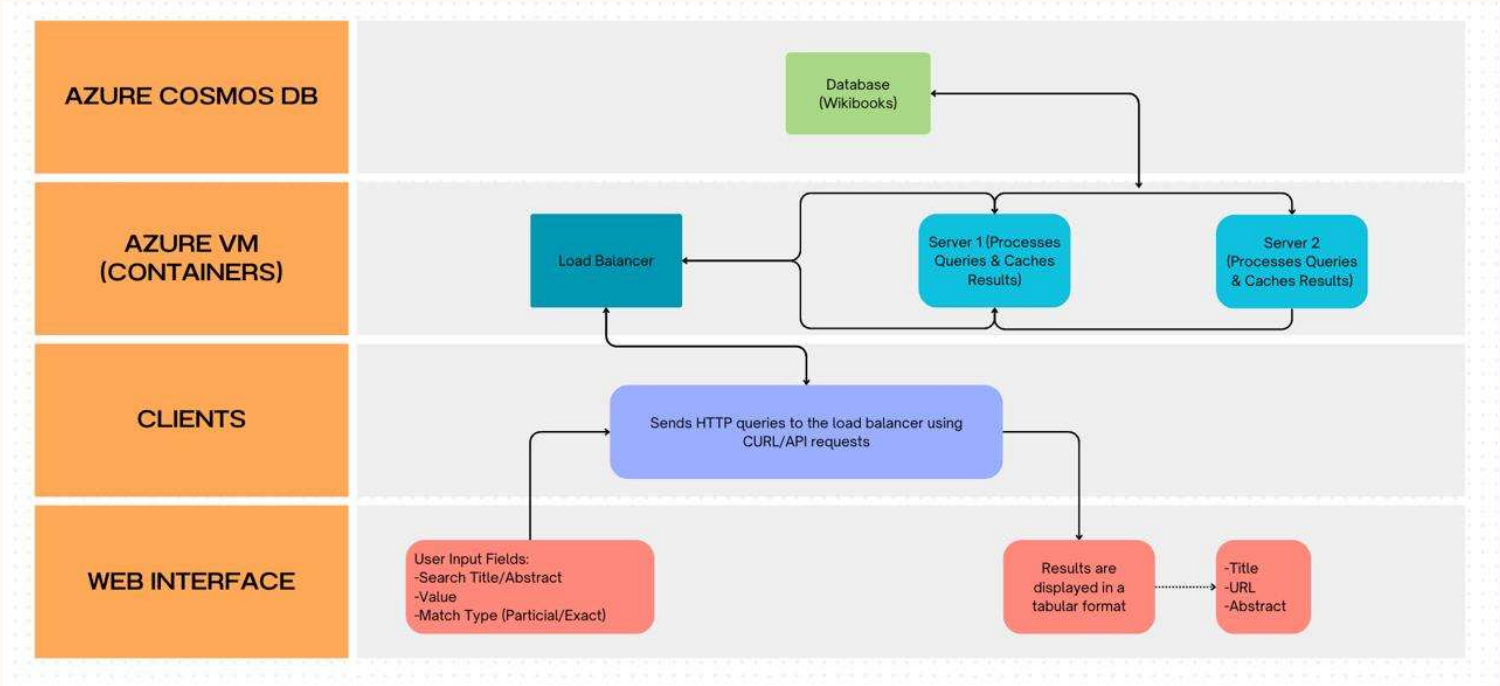
In the era of big data, efficient processing of large datasets is essential for modern applications. Distributed systems provide scalable and reliable solutions by leveraging multiple nodes to share workloads and improve system performance. This project presents a distributed system designed to query and process a 3 GB WikiBooks dataset, integrating key principles of **containerization**, **concurrency**, and **caching** to ensure high performance and fault tolerance.

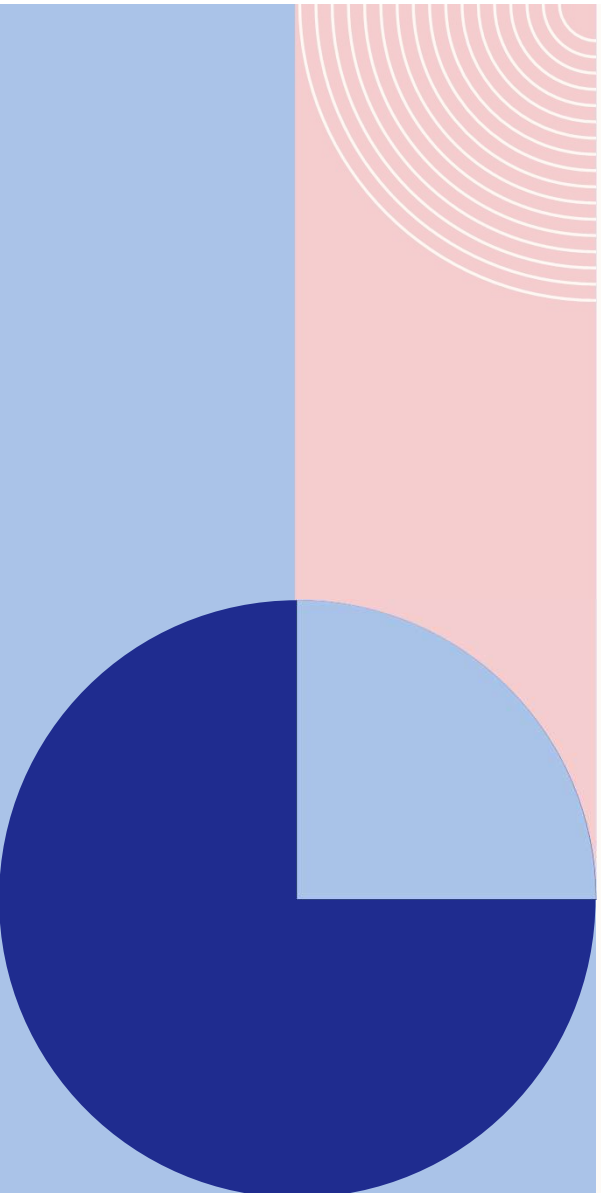
- The system consists of a **load balancer**, multiple **server instances**, and a **database**. The load balancer dynamically distributes client queries to the least busy server, optimizing resource utilization. Server instances, hosted in Docker containers, process queries, interact with Azure Cosmos DB for data retrieval, and utilize Redis caching to minimize redundant queries and improve response times.

- To handle concurrent queries efficiently, a **queue-based asynchronous processing model** is implemented, enabling the system to process multiple requests in parallel while maintaining high throughput. The containerized architecture ensures scalability, allowing the system to add or remove server instances as needed to accommodate varying workloads.

- A user-friendly **web interface** enables seamless query submission and result visualization, abstracting backend complexities. By combining modern distributed system design, cloud-based storage, and caching techniques, the project delivers a robust, scalable, and resource-efficient solution for large-scale data processing challenges.

SYSTEM ARCHITECTURE





- A **distributed architecture** hosted on an Azure Virtual Machine (VM). Azure Cosmos DB serves as the centralized database, ensuring scalable and low-latency data retrieval.

- A **load balancer** dynamically distributes client queries to two server containers based on their workload, optimizing resource usage. The servers process queries, interact with Cosmos DB, and utilize **Redis caching** to improve performance by reducing redundant database calls. Clients submit queries via a **web interface**, which abstracts backend complexity and displays results—such as Title, URL, and Abstract—in a user-friendly tabular format.

- The system combines containerization, concurrency, and caching to deliver a **scalable, fault-tolerant**, and **high-performance** solution for large-scale data processing.

Implementation Steps:

1. Setting up the Project Environment:

Azure VM Setup:

- ❑ Provision an **Azure Virtual Machine (VM)** to host the distributed system components.
- ❑ Choose an appropriate VM instance type based on resource needs (e.g., CPU, memory).
- ❑ Install required software packages on the VM:
  - **Docker** for containerization.
  - **Python3**, Flask, Redis, and Azure Cosmos DB SDK for the backend.

2. Database Configuration (Azure Cosmos DB)

- ❑ **Provision Cosmos DB:**
  - •Create an **Azure Cosmos DB account** in the Azure portal.
  - •Set up a new database
- ❑ **Import the Dataset:**
  - •Prepare the **3 GB WikiBooks dataset** for ingestion into Cosmos DB.
- ❑ **Configure Access Credentials:**
  - •Generate the **endpoint URL** and **primary key** for Cosmos DB.

3. Backend Implementation

Load Balancer (Forward Request)

1.Purpose:

- To manage incoming client requests and distribute them to the least loaded server.

2.Implementation Steps:

- Use **Flask** to create an API endpoint for receiving client requests.
- Implement a **dynamic load distribution** mechanism:
  - Maintain a dictionary to track server connections.
  - Identify the server with the least number of connections.
- Forward the query request to the selected server using the requests library.
- Once the server processes the query, collect the response and return it to the client.

SERVERS (PROCESS QUERIES AND CACHING)

❑ **Purpose:**

To process queries, fetch data from Cosmos DB, and cache results in Redis.

❑ **Implementation Steps:**

1.Set up a **Flask-based API server** with endpoints for query processing.
2.Use **worker threads** to process queries asynchronously using a queue.
3.Implement a **caching mechanism** with Redis:

- ✓ Check the cache for existing results.
- ✓ If the result is **not cached** (cache miss), query Cosmos DB and cache the result.
- ✓ If the result is **cached** (cache hit), retrieve the data from Redis.

3.SERVER QUEUE AND WORKER THREADS:

USE THE MODULE QUEUE.QUEUE TO HANDLE ASYNCHRONOUS QUERY PROCESSING

CREATE WORKER THREADS TO FETCH QUERIES FROM THE QUEUE AND PROCESS THEM.

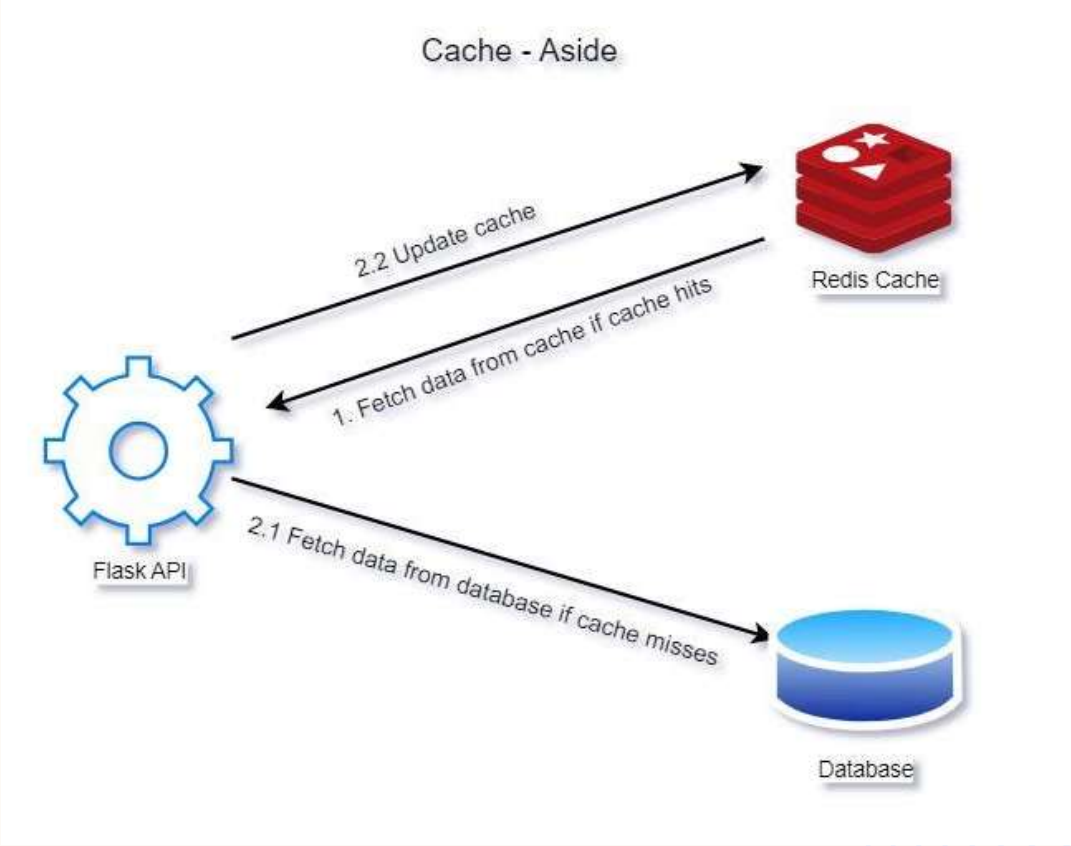
Caching Workflow:

Generate a **unique cache key** based on the query parameters.

Store the query results in Redis with a **TTL** (**Time-to-Live**) to

prevent stale data

.



4. Containerization with Docker

1.Dockerize the Load Balancer and Servers:

Create a docker file to package the flask servers and dependencies

2.Docker Compose Configuration:

Use docker-compose.yml to orchestrate the containers

*One container for the **Load Balancer**.

*Two containers for the **Servers**.

3.Run the Containers

.

5. CLIENT IMPLEMENTATION

1.Command- line Queries :

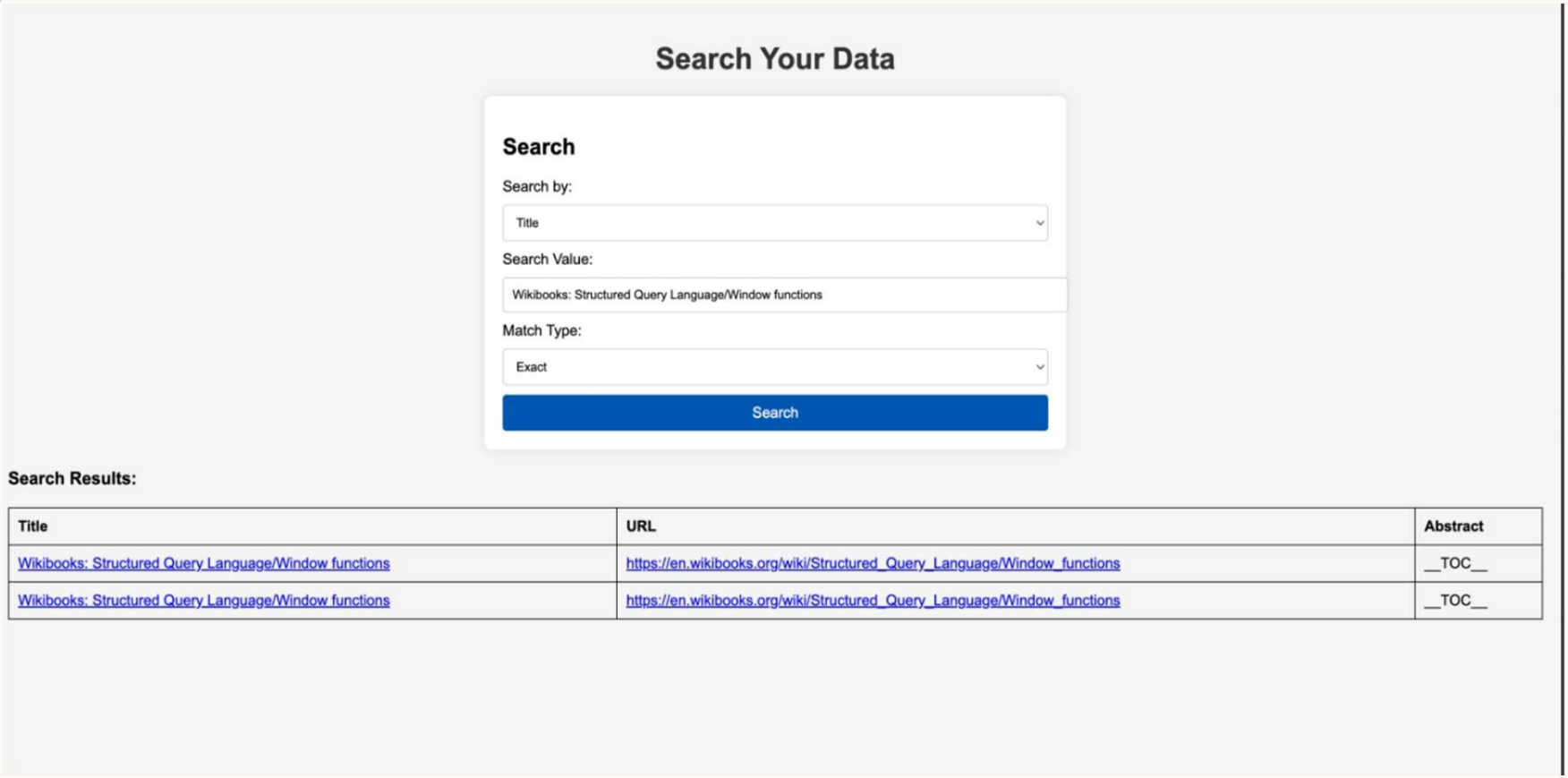
Use tools like CURL to test the system.

2.Web Interface:

Create an interactive HTML-based web interface for query submission.

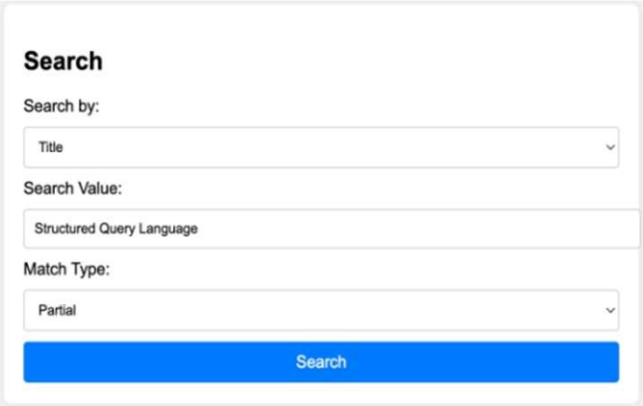
Use **JavaScript** to send HTTP requests to the load balancer.

Dynamically display the query results in a **tabular format**.



Search Results:

Title	URL	Abstract
Wikibooks: Structured Query Language/Window functions	https://en.wikibooks.org/wiki/Structured_Query_Language/Window_functions	__TOC__
Wikibooks: Structured Query Language/Window functions	https://en.wikibooks.org/wiki/Structured_Query_Language/Window_functions	__TOC__



Search Results:

Title	URL	Abstract
Wikibooks: Structured Query Language/Window functions	https://en.wikibooks.org/wiki/Structured_Query_Language/Window_functions	__TOC__
Wikibooks: Structured Query Language/Language Elements	https://en.wikibooks.org/wiki/Structured_Query_Language/Language_Elements	SQL consists of statements that start with a keyword like SELECT, DELETE or CREATE and terminate with a semicolon. Their elements are case-insensitive except for fixed character string values like 'Mr.
Wikibooks: Structured Query Language/Example Database Data	https://en.wikibooks.org/wiki/Structured_Query_Language/Example_Database_Data	__TOC__
Wikibooks: Structured Query Language/Relational Databases	https://en.wikibooks.org/wiki/Structured_Query_Language/Relational_Databases	Before learning SQL, relational databases have several concepts that are important to learn first. Databases store the data of an information system.
Wikibooks: Structured Query Language/Data Types	https://en.wikibooks.org/wiki/Structured_Query_Language/Data_Types	__TOC__
Wikibooks: Structured Query Language/Managing Indexes	https://en.wikibooks.org/wiki/Structured_Query_Language/Managing_Indexes	__TOC__
Wikibooks: A-level Computing/AQA/Paper	https://en.wikibooks.org/wiki/A-	

RESULTS

Performance with and without Caching

- Caching significantly enhances system performance by reducing the need to query **Azure Cosmos DB** repeatedly. Results for frequently accessed queries are stored in **Redis**, allowing servers to fetch them directly from the cache instead of querying the database.
- The performance comparison between **with caching** and **without caching** highlights the impact of this approach:

Without Caching:

The query required direct access to Azure Cosmos DB, resulting in an average response time of **3.65 seconds** (measured as **3.654987096786499 seconds**). This is due to the time required for data retrieval and network latency.

- **With Caching:**
  When the query result was already cached in Redis, the response time was drastically reduced to **0.00023 seconds** (measured as **0.00023150444030761 72 seconds**). This highlights the efficiency of serving results directly from the cache.

- This comparison demonstrates a **significant improvement** in query response time when caching is implemented. By reducing latency and optimizing database usage, caching enhances the overall system performance, particularly for frequently repeated queries.
- The reduction from **3.65 seconds** to **0.00023 seconds** underscores the importance of Redis in minimizing query overhead and improving user experience.

LIMITATIONS AND FUTURE WORK:

1. Single Point of Failure for Load Balancer
2. Manual Scaling
3. Limited Fault Tolerance
4. Cache Invalidation
5. Network Latency
6. Lack of Monitoring and Analytics

CONCLUSIONS

Overall, the project demonstrates the ability to process large-scale data efficiently in a distributed environment. By leveraging load balancing, caching, and containerization, the system achieves robustness, scalability, and optimized performance. Future enhancements will ensure the system is even more fault-tolerant, flexible, and capable of handling higher workloads, making it a robust solution for real-time query processing and data management.